The Lifecycle of Semantic Parsing
From Template-based Data Collection to Neural Decomposable Parsers

Anonymous EMNLP submission

Abstract

Scaling semantic parsing to arbitrary domains faces two interrelated challenges. Collecting compositional training data (i.e., query-logical form pairs), quickly and cheaply, and building semantic parsers which can effectively capture compositionality. We address these challenges by presenting a framework which allows to efficiently elicit compositional datasets and an interpretable neural parser which generates derivation trees of logical forms. The key idea is to treat a logical form as a compositional sequence of templates, where each template corresponds to a decomposed fragment of the logical form. We ask annotators to summarize a template sequence into a natural language query, thereby obtaining parallel triples of queries, logical forms, and their decompositions. We crowdsource a dataset covering six domains and train a neural parser which outperforms baselines with a significant margin.[1]

1 Introduction

Semantic parsing is the task of converting natural language queries into machine interpretable meaning representations which can be executed against a real-world environment such as a database. For example, in the restaurant booking domain, the query "*which restaurant serves Thai food and is nearest to me*" can be parsed into the logical form argmin(food_type(Thai food), distance) which when executed against a database will return a restaurant name.

Approaches to learning semantic parsers have for the most part used training data consisting of queries paired with logical forms. But labeling such data requires expert knowledge, e.g., familiarity with logical forms and the domain at hand, rendering the task expensive and difficult to scale. Wang et al. (2015) advocate crowd-sourcing as a means of mitigating the paucity of training data for new domains. The basic idea is to use a

[1]We release our dataset, semantic parser, and data collection method at http://www.xxx.yyy.

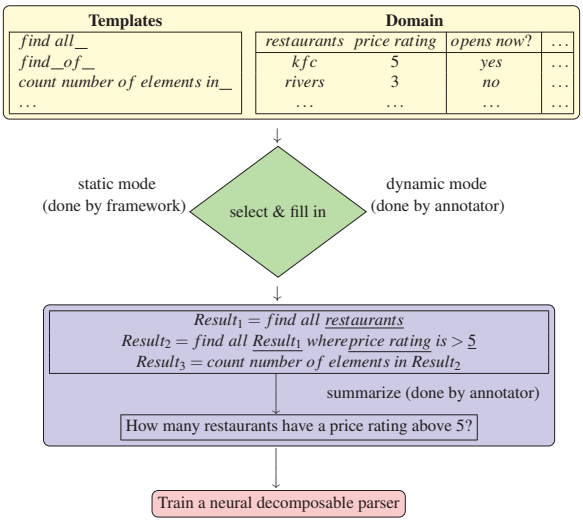

Figure 1: An overview of our framework.

synchronous grammar to generate logical forms paired with artificial queries which crowdworkers are asked to paraphrase. For example, the logical form argmin(food_type(Thai food), distance) is deterministically mapped to "*restaurants with minimum distance with food type thai*", which will be later paraphrased by crowdworkers into more naturally sounding English (e.g., "*nearest restaurants serving thai food*"). Since the readability of auto generated queries decreases when the complexity of the the querying task and its underlying logical form increases, the approach primarily targets utterances exhibiting limited compositionality often with two predicates and entities.

Our first contribution in this work is to extend the approach of Wang et al. (2015) to handle more complicated querying tasks. Instead of representing a logical form with a single artificial sentence, our key insight is to represent it as a sequence of templates where each template corresponds to a fragment of the logical form. And so the query argmin(food_type(Thai food), distance) would be represented with the templates "*find the restaurants that serve thai food*" and "*of those, find the clos-*

est". Iyyer et al. (2017) show that when expressing a complex querying task, users often employ a set of inter-related short sentences; such decomposed queries can potentially handle higher levels of compositionality. Since templates correspond to specific aspects of the logical form, they are easier to understand by crowdworkers than a more elaborate auto-generated query representing the entire logical form. We consider two modes of data elicitation (see Figure 1), striking a balance between efficiency and flexibility. In the *static* mode, our framework first generates a logical form in the backend, and then looks up and fills in natural language templates corresponding to the rules used to derive the logical form. Annotators are then shown the filled templates and are asked to write down a natural language query that summarizes the querying task. In the *dynamic* mode annotators are given more flexibility: they can select the templates, fill them, and produce a valid query. This gives them freedom to write queries that can be anticipated in a given domain. We adopt the static mode to collect a dataset of six domains consisting of 7,708 query-logical form pairs. We also experiment with the dynamic mode and discuss the type of the data it yields.

Our second contribution is to leverage the annotations elicited via our framework to build an interpretable neural semantic parser. Several models have been developed over the years to learn semantic parsers from queries paired with logical forms. Early work has mainly focused on chart parsers with hand-crafted features and dynamic programming inference (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2011; Berant et al., 2013). While more recently neural sequence-to-sequence models (Dong and Lapata, 2016; Jia and Liang, 2016; Suhr et al., 2018) have shown promising results. These models generate logical forms as left-to-right strings. We differ from this work in that we train a neural semantic parser that generates derivation trees of logical forms. Our model shares similarities with more recent neural semantic parsers that predict grammar rules sequentially (Krishnamurthy et al., 2017; Yin and Neubig, 2017), but our decoder is more structured and it outputs derivation trees directly. This is made possible by our data elicitation framework which caches (in the form of templates) the decomposition and derivation for each logical form (i.e., the logical form fragments and the way they are combined to derive the target logical form). When tested on our six domains, our parser outperforms existing sequence-to-sequence models with a significant margin. We also demonstrate additional improvements with cross-domain training.

2 Data Collection Module

In this section we describe our data collection approach. We first outline how logical forms are built and then discuss the two modes of data elicitation supported in the framework.

2.1 Domain General and Specific Aspects

Throughout this paper, logical forms are constructed with the same programming language which Wang et al. (2015) adopted to instantiate lambda DCS (Liang et al., 2011). The latter is a simple yet expressive logical language where composition operates over sets (of entities) rather than truth values. However, our framework can also be extended to other semantic formalisms (e.g., lambda calculus).

We follow Wang et al. (2015) in distinguishing domain-general from domain-specific aspects of logical forms. Domain-general aspects are logical rules or operations stemming from the logical language used by the semantic parser. In our work they include lambda expressions which specify functionalities such as counting, aggregation, and filtering. Table 1 (second column) shows examples of such expressions (see the supplementary material for a detailed specification of the logical formulation). Domain-specific aspects refer to seed lexicons that map domain-specific predicates or entities to their natural language descriptions. For example, Table 2 shows a seed lexicon for the restaurant domain which covers binary predicates (e.g., custom_rating), unary predicates (e.g., open_now), entities (e.g., restaurant.kfc), and their natural language descriptions. Note that natural language descriptions are important in cases where domain-specific predicates or entities are not verbalized (for example a predicate may be simply represented as an index m.001 in the database). Such descriptions must be provided to annotators to allow for basic understanding, to enable the paraphrasing task.

2.2 Annotation Modes

Our goal is to extend the work of Wang et al. (2015) with a more intuitive interface for crowdworkers to annotate data representing more complex querying tasks. Our central idea is to map each domain-general rule onto a natural language

ID	Domain-general rules	Natural language templates
LookupKey	λs:(call getProperty (var s))	*find all $s*
LookupValue	λpλs:(call getProperty (var s) (var p))	*find $p of $s*
Filter(property)	λsλpλv:(call filter (var s) (var p) (string =) (var v))	*find $s where $p is $v*
Filter(assertion)	λsλp:(call filter(var s) (var p))	*find $s which satisfies $p*
Count	λs:(call size (var s))	*count number of elements in $s*
Sum	λs:(call sum (var s))	*sum all elements in $s*
Comparative(<)	λsλpλv:(call filter (var s) (var p) (string <) (var v))	*find $s with $p < $v*
CountComparative(<)	λsλpλv:(call countComparative (var s) (var p) (string <) (var v))	*find $s with number of $p < $v*
Superlative(min)	λsλp:(call superlative (var s) (string min) (var p))	*find $s with smallest $p*
CountSuperlative(min)	λsλp:(call countSuperlative (var s) (string min) (var p))	*find $s with smallest number of $p*

Table 1: Domain-general rules and their natural language templates. Each rule is specified by the logical language of the semantic parser and implemented as a function at the back-end. Templates are specified by our framework and used by annotators at the front-end. Comparative and CountComparative have four variants ($<$, $>$, $\leq$ or $\geq$), while only one is shown. Superlative and CountSuperlative have two variants (min, max).

ID	Domain-specific aspects	Descriptions
BinaryPredicate	custom_rating	*custom rating*
	price_rating	*price rating*
UnaryPredicate	open_now	*open now*
	take_away	*offer take away*
	delivery	*offer delivery*
Entity	restaurant.kfc	*kfc*

Table 2: Domain-specific aspects for the restaurant domain (provided by domain experts): binary predicates (properties), unary predicates (assertions), entities, and their natural language descriptions.

template which acts as a front-end presented to annotators. Table 1 (third column) displays a collection of templates underlying those rules. Templates are described in such a way that can be understood by annotators who have no knowledge of the logical language. As shown in Figure 1 our framework supports two modes of data collection. Examples of the two modes are shown in Tables 3 and 4, respectively.

Static Mode Once a domain manager specifies the necessary domain-specific information (e.g., restaurant names and properties), our framework obtains query-logical form pairs in the following steps:

1. It generates a context-free logical form following the grammar of the logical language. The logical form can be of arbitrary compositionality, and the rules used to derive it (i.e., decompositions) are cached.
2. It finds the domain-general rules in the derivation, and looks up the corresponding templates which are then instantiated with domain-specific rules; recall that domain-specific predicates and entities are replaced by their natural language descriptions.
3. The templates are displayed to crowdworkers, who are asked to write down a query that summarizes the querying task described by the templates. The summary can consist of a few short sentences too.

Dynamic Mode The dynamic mode offers annotators the flexibility to create both queries and logical forms on their own. We envisage the primary users of our framework in this mode being domain managers who have a better understanding of (or can devote more time to) the domain-specific task. As in the static mode, domain managers must initially specify a lexicon. Next, query-logical form pairs are collected as follows:

1. The framework displays all available templates and a table containing domain information to the annotator. The table includes natural descriptions of all available predicates. However, the list of entities need not be exhaustive, since it is straightforward to apply entity replacement (for entities of the same type) to construct more query-logical forms from a set of base query-logical forms.
2. The annotator manually selects and fills in a sequence of templates to come up with a querying task. In this process, logical rules are applied recursively at the back-end to construct the final logical form.
3. The annotator writes down a query that summarizes the querying task and obtains a query-logical form pair.

The two modes of annotation allow to trade efficiency with flexibility. The static mode only requires annotators to summarize a collection of mature templates to create datasets quickly. The dynamic mode is more expensive but offers annotators the flexibility to use templates to create their own querying tasks.

1. Using the following logical form as an example: (call listValue (call size (call filter (call filter (call getProperty(singleton type.restaurant) (string !type)) (rel.distance) (string <) (num.500)) (rel.cuisine) (string =) cuisine.thai))) Logical forms are generated **automatically** by recursively applying domain-general and domain-specific rules: Result_1=λs:(call getProperty (var s)), s=type.restaurant Result_2=λsλpλv:(call filter (var s) (var p) (string <) (var v)), s=Result_1, p=rel.distance, v=num.500 Result_3=λsλpλv:(cal filter (var s) (var p) (string =) (var v)), s=Result_2, p=rel.cuisine, v=cuisine.thai Result_4=λs:(call size (var s)), s=Result_3
2. Templates corresponding to domain-general rules are retrieved and filled with domain-specific details (shown in brackets): *Result_1 = find all* [*restaurants*] *Result_2 = find* [Result_1] *at* [*distance*] < [*500*] *Result_3 = find* [Result_2] *where* [*cuisine*] *is* [*Thai*] *Result_4 = count number of* [Result_3]
3. The filled templates are displayed to the annotator, whose job is to write down a query summarizing them: *How many restaurants within 500m serve Thai food?*

Table 3: An example of the static data collection mode in the restaurant domain. Annotators write down a query that summarizes the task described by a collection of templates.

1. Templates (unfilled) from Table 1 and domain information (see Table 2) are displayed to the annotator.
2. The annotator **manually** selects and fills in a few templates to come up with a querying task: *Result_1 = find all* [*restaurants*] *Result_2 = find* [Result_1] *at* [*distance*] < [*500*] *Result_3 = find* [Result_2] *where* [*cuisine*] *is* [*Thai*] *Result_4 = count number of* [Result_3]
3. Domain-general/specific rules underlying these templates are retrieved and recursively applied: Result_1=λs:(call getProperty (var s)), s=type.restaurant Result_2=λsλpλv:(call filter (var s) (var p) (string <) (var v)), s=Result_1, p=rel.distance, v=num.500 Result_3=λsλpλv:(call filter (var s) (var p) (string =) (var v)), s=Result_2, p=rel.cuisine, v=cuisine.thai Result_4=λs:(call size (var s)), s=Result_3 The following logical form is generated: (call listValue (call size (call filter (call filter (call getProperty(singleton type.restaurant) (string !type)) (rel.distance) (string <) (num.500)) (rel.cuisine) (string =) cuisine.thai)))
4. The annotator writes down a query that summarizes the querying task they have come up with. *How many restaurants within 500m serve Thai food?*

Table 4: An example of the dynamic data collection mode in the restaurant domain. Annotators come up with their own querying task using templates, and then write down a query summarizing it.

3 Neural Semantic Parsing Module

Another advantage of our framework stems from the fact that it caches the (sequence of) logical rules used to obtain the final logical form. Most existing semantic parsing datasets contain query-logical form pairs, while the derivations[2] of logical forms are latent. In such cases, a grammar induction step is needed to infer the derivations during training (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2011), and often the recovery of derivations from logical forms is ambiguous (Kwiatkowksi et al., 2010). Recent neural sequence-to-sequence models do not consider any grammar (Dong and Lapata, 2016; Jia and Liang, 2016), but operate at the string level instead. In contrast, our annotation method allows us to build a neural parser which is both interpretable and decomposable, i.e., it generates the derivation trees of logical forms step by step, as we explain below.

[2]A derivation refers to the sequence of logical rules applied to obtain a logical form.

Overview We encode the input query with a bidirectional LSTM and generate the derivation tree of the logical form with a stack-LSTM (Dyer et al., 2015). In the generation process, a sequence of logical rules is predicted following the pre-order traversal of the derivation tree. For example, in the restaurant domain, the space of all logical rules consists of the general rules in Table 1 and the domain-specific ones in Table 2. For simplicity, we denote each rule with a functional operator as shown in the tables. For domain-specific rules, we first predict their general category (binary predicate, unary predicate, or entity) and then the specific predicate or entity choice (with different neural network parameters). The resulting space of rule predictions is [LookupKey, LookupValue, Filter, Count, Sum, Comparative, CountComparative, Superlative, CountSuperlative, BinaryPredicate, UnaryPredicate, Entity]. The derivation tree for the logical form (rendered in red) in Table 3 is shown in Figure 2.

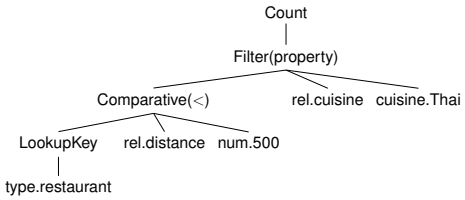

Figure 2: Derivation tree for the logical form (shown in red) from Table 3. Domain-general rules are represented as non-terminal nodes, using the abbreviations shown in Table 1. For example, Count refers to λs:(call size (var s)). Domain-specific aspects are represent as terminal nodes.

Encoder We encode the query x with a bidirectional LSTM which processes a variable-length query $x = (x_1, \cdots, x_n)$ into a list of token representations $[h_1, \cdots, h_n]$, where each representation is the concatenation of the corresponding forward and backward LSTM states.

Decoder After the query is encoded, we generate the derivation tree of the logical form with a stack-LSTM decoder (Dyer et al., 2015). As shown in Figure 2, non-terminal nodes of the derivation tree are domain-general rules, while terminal nodes are domain-specific ones. Advantageously, the stack-LSTM is able to capture both the partially-completed tree structure, and the rules applied. When a non-terminal or terminal rule y_t is newly predicted, the stack-LSTM state, denoted by g_t, is updated from its older state g_{t-1} as an ordinary LSTM:

$$g_t = \text{LSTM}(y_t, g_{t-1}) \quad (1)$$

The new state is additionally pushed onto the stack marking whether it corresponds to a non-terminal or terminal.

The generation of the tree nodes is performed in pre-order and the model needs to identify when the prediction of subtree branch is completed—then a beta reduction step should be applied to the corresponding logical rule. To achieve that, we rely on the built-in "reduce" mechanism of stack-LSTM and incorporate Reduce as an additional rule in the rule prediction space: When a subtree branch is "reduced" (or completely predicted), the states of the stack-LSTM are recursively popped from the stack until a non-terminal is encountered. This non-terminal state is popped as well, after which the stack-LSTM reaches an intermediate state denoted by $g_{t-1:t}$. At this point, we compute the representation of the completed subtree z_t as:

$$z_t = W_z \cdot [p_z : c_z] \quad (2)$$

where p_z denotes the parent (non-terminal) embedding of the subtree, and c_z denotes the average embedding of the children (terminals or already-completed subtrees). W_z is the weight matrix. Finally, z_t serves as input for updating $g_{t-1:t}$ to g_t:

$$g_t = \text{LSTM}(z_t, g_{t-1:t}) \quad (3)$$

Prediction At each time step of the decoding, the parser first predicts a rule conditioned on the decoder state g_t and the encoder states $h_1 \cdots h_n$. We apply standard soft attention between g_t and the encoder states $h_1 \cdots h_n$ to compute a feature representation $\bar{h}_t$:

$$u_t^i = V \tanh(W_h h_i + W_g g_t) \quad (4)$$

$$a_t^i = \text{softmax}(u_t^i) \quad (5)$$

$$\bar{h}_t = \sum_i a_t^i h_i \quad (6)$$

where V, W_h and W_g are all weight parameters. The prediction of the next rule y_{t+1} is computed with a softmax classifier, which takes the concatenated features $\bar{h}_t$ and g_t as input:

$$y_{t+1} \sim \text{softmax}(W_y \tanh(W_f[\bar{h}_t, g_t])) \quad (7)$$

Recall that for domain-specific (terminal) rules, our parser first predicts their high-level category (binary/unary predicate or entity). Only when y_{t+1} matches one of the terminal categories, we further predict a fine-grained predicate or entity, with another set of neural parameters:

$$y_{t+1} \sim \text{softmax}(W_{y'} \tanh(W_{f'}[\bar{h}_t, g_t])) \quad (8)$$

where W_y and $W_{y'}$ are different weight matrices.

4 Experiments

In this section, we first describe how we used our data elicitation framework to obtain annotations for six domains in the static mode. We then discuss experiments on this dataset with the semantic parser just described. We finally compare our framework to Wang et al. (2015), and provide evaluation of the dynamic mode.

4.1 Data Collection in Static Mode

Our static-mode data collection is concentrated on six domains, two of which relate to company management (meeting and employees databases), two concern recommendation engines (hotel and restaurant databases), and two target healthcare applications (disease and medication databases).

#rules	1				2				3				4				All			
Domain	Q	Tp	Tk	WO	Q	Tp	Tk	WO	Q	Tp	Tk	WO	Q	Tp	Tk	WO	Q	Tp	Tk	WO
meeting	378	191	9.1	1.41	570	537	16.20	2.21	254	254	16.2	2.81	46	46	21.37	3.19	1,248	1,028	12.59	2.21
employees	322	240	8.96	1.34	320	319	13.47	2.95	486	486	18.2	3.66	268	268	22.37	3.12	1,396	1,313	15.79	3.12
hotel	170	146	8.99	1.91	358	542	16.86	3.64	542	542	16.86	4.11	433	433	19.89	4.05	1,503	1,479	15.87	4.05
restaurant	132	98	8.14	1.40	301	295	12.74	3.41	495	495	16.74	3.74	311	311	20.02	3.66	1,239	1,199	15.68	3.66
disease	283	212	9.3	1.29	301	455	14.23	3.52	455	455	19.49	5.65	213	213	23.95	4.48	1,252	1,176	16.68	4.48
medication	136	102	8.53	1.31	252	246	11.89	2.35	435	435	16.09	3.17	247	247	20.04	2.91	1,070	1,030	15.05	2.91

Table 5: Number of queries (Q), number of templates (Tp), average number of tokens (Tk) and token overlap between queries and templates (WO) per domain and overall (All).

The dataset was collected with Amazon Mechanical Turk (AMT). Across domains, the total number of querying tasks (described by templates) that the framework generated was 7,225. These tasks were sampled randomly (without replacement) to show to crowdworkers. Each worker saw three tasks per HIT and was paid 0.3$. After removing repeated query-logical form pairs, we collected a semantic parsing dataset of 7,708 examples. The average amount of time annotators spent on each domain was three hours. We evaluated the correctness of 100 randomly chosen queries, and the accuracy was 81%. We did not conduct any manual post-processing as our aim was to simulate a real-world scenario that handles noisy data.

Table 5 shows the number of queries (Q) we obtained for each domain broken down according to the depth of compositionality. The table also provides statistics on the number of templates (Tp) per domain, the average number of tokens (Tk) per query in each domain, and the token overlap (Ov) between the queries and the corresponding templates. Overall, we observe that the dataset reveals a high degree of compositionality across domains. The number of queries collected at each compositional level is dependent on the space of predicates in each domain. We also see that query length does not vary drastically among domains even though the average number of tokens is affected by the verbosity of entities and predicates in each domain. We use word overlap as a measure of the amount of paraphrasing. We see that queries in the disease domain deviate least from their corresponding templates while queries in the meeting domain deviate most. This number is affected by the degree of expert knowledge required for paraphrasing in each domain.

4.2 Semantic Parsing Results

In the process of building the above dataset, our framework cached the derivations of logical forms (rules or templates used in sequence). This allowed us to train a neural semantic parser that generates the logical form by following its derivation, as explained in Section 3.

In our experiments, all LSTMs had one layer with 150 dimensions. The word embedding size and the rule embedding size were set to 50. A dropout of 0.5 was used on the input features of the softmax classifiers in Equations (7) and (8). Momentum SGD was used to update the parameters of the model. We implemented two baselines, a sequence-to-sequence parser (Dong and Lapata, 2016; Jia and Liang, 2016) and a sequence-to-tree parser which follows peripheral tree-structured constrains of the logical form (Cheng et al., 2017). To give a concrete example, for the logical form in Table 3 (shown in red), the sequence-to-sequence baseline predicts the entire string from left to right, including auxiliary tokens (and). The sequence-to-tree baseline predicts the logical form as a tree (but not the corresponding derivation tree), where brackets (and) are not predicted but rather used as cues to recover the tree structure (e.g., the token call is predicted as a non-terminal).

Table 6 shows results on the test set of each domain using exact match as the evaluation metric (ExM). Our sequence-to-derivation tree model (S2D) yields substantial gains over the baselines (S2S and S2T) across domains. However, a limitation of exact match is that different logical forms may be equivalent to the commutativity and associativity of rule applications (Xu et al., 2017). For example, two subsequent Filter rules in a logical form are interchangeable. For this reason, we additionally compute the number of logical forms that match the gold standard at the semantic level (see SeM in Table 6). This is done by enumerating all variants of each gold standard logical form and checking if the corresponding prediction matches any of them. Again, we find that the sequence to derivation tree model outperforms related baselines by a wide margin.

We conducted further experiments by training a single model on data from all domains, and testing it on the test set of each individual domain. As the results in Table 7 reveal, we obtain gains for most domains on both metrics of ex-

Model	meeting		employees		hotel		restaurant		disease		medication	
	ExM	SeM	ExM	SeM	ExM	SeM	ExM	SeM	ExM	SeM	ExM	SeM
S2S	37.2	43.2	14.3	17.5	24.5	31.2	21.3	29.0	16.7	23.2	15.5	16.7
S2T	41.2	46.8	21.4	28.2	31.5	43.5	25.4	35.9	22.4	34.4	28.2	33.9
S2D	45.6	54.0	27.8	35.5	39.2	52.6	47.2	49.5	26.9	44.6	35.8	46.2

Table 6: Perfomance on various domains (test set) using exact match (ExM) and semantic match (SeM).

Domain	ExM	SeM
meeting	(45.6) 48.8	(54.0) 56.8
employees	(27.8) 31.7	(35.5) 41.4
hotel	(39.2) 41.8	(52.6) 56.1
restaurant	(47.2) 33.1	(49.5) 48.8
disease	(26.9) 30.7	(44.6) 48.3
medication	(35.8) 37.4	(46.2) 51.8

Table 7: Sequence-to-derivation tree model (S2D) trained on *six* domains and evaluated on the test set of each domain. Results of S2D when trained and tested on a *single* domain are shown in parens.

act and semantic match. Our results agree with previous work (Herzig and Berant, 2017) which improves semantic parsing accuracy by training a single sequence to sequence model over multiple knowledge bases. Since domain-general aspects are shared, when training across domains, the parser receives more supervision cues on discovering these domain-general aspects from their natural language descriptions.

4.3 Comparison to Wang et al. (2015)

Our static mode is similar to Wang et al. (2015) in that we also ask crowdworkers to paraphrase artificial expressions into natural ones. In their approach crowdworkers paraphrase a *single* artificial sentence, whereas in our case they are asked to paraphrase a *sequence* of templates. We directly evaluated how crowdworkers perceive our templates compared to single sentences. For each domain we randomly sampled 24 querying tasks described by templates (144 tasks in total) and derived the corresponding artificial language using Wang et al.'s (2015) grammar. Artificial sentences and templates were also paired with a natural language description (generated by us) which explained the task (see Table 10 for examples). Workers were asked to rate how well the sentence and templates corresponded to the natural language description according to two criteria: (a) intelligibility (how easy is the artificial language to understand?) and (b) accuracy (does it match the intention of the task?). Participants used a 1–5 rating scale where 1 is worst and 5 is best. We elicited 5 responses per task.

Table 8 summarizes the mean ratings for each domain and overall. As can be seen, our approach

Domain	Intelligibility		Accuracy		Combined	
	S	T	S	T	S	T
meeting	3.54	3.81	3.86	4.02	3.70	3.91
employee	3.58	4.13	3.48	4.43	3.53	4.28
hotel	3.81	3.96	3.79	4.29	4.12	3.80
restaurant	3.93	4.23	3.80	3.75	3.86	4.13
disease	3.41	3.69	3.68	4.02	3.55	3.86
medication	3.83	3.97	3.83	4.40	3.83	4.19
All	3.96	3.67	3.55	4.03	3.70	4.06

Table 8: Comparison between artificial sentences (S; Wang et al., 2015) and template-based approach (T). Mean ratings are shown per domain and overall. Combined is the average of Intelligibility and Accuracy. Means are underlined if their difference is statistically significant at $p < 0.05$ using a post-hoc Turk test.

Depth	Intelligibility		Accuracy		Combined	
	S	T	S	T	S	T
1	4.00	4.29	4.05	4.29	4.03	4.26
2	4.11	4.18	4.03	4.18	4.07	4.02
3	3.61	4.09	3.60	4.01	3.58	4.01
4	3.56	4.28	3.35	4.25	3.60	4.10

Table 9: Comparison between artificial sentences (S; Wang et al., 2015) and template-based approach (T) for varying compositionality depths. Mean ratings are aggregated across domains. Combined is the average of Intelligibility and Accuracy. Means are underlined if their difference is statistically significant at $p < 0.01$ using a post-hoc Turk test.

generally receives higher ratings for Intelligibility and Accuracy. When both types of ratings our combined the templates significantly outperform the individual sentences for all domains but meetings. Table 9 shows a breakdown of our results according to the depth of compositionality. Queries of depth 1 and 2 can be easily described by one sentence, and our template-based approach has no clear advantage over Wang et al. (2015). However, when the compositional depth increases to 3 and 4, templates are perceived as more intelligible and accurate across domains; all means differences for depths 3 and 4 are statistically significant ($p < 0.01$). Further qualitative analysis suggests that our approach receives higher ratings in cases where the output of the grammar from Wang et al. (2015) involves various propositional attachment ambiguities. The ambiguities are common when the compositional depth increases (Example 1 in Table 10), when the query contains conjunc-

Task description	Our templates	Wang et al. (2015)
We have a database of diseases and would like to find diseases which have fever as their symptom. These diseases should be treatable with antibiotics. Their incubation period is longer than a day. If you have such a disease you should see a doctor.	R_1 = *find the diseases whose symptom is fever* R_2= *find* R_1 *whose treatment is antibiotics* R_3= *find* R_2 *whose incubation period is longer than a day* *QR= find* R_3 *which require to see a doctor*	diseases whose symptom is fever whose treatment is aspirin whose incubation period is larger than a day which require to see a doctor
We have a database of diseases and would like to find diseases which have fever as their symptom; amongst them, we would like to find those with heart disease as complication. Finally, we want to find all diseases that can be treated with antibiotics.	R_1= *find the diseases whose symptom is fever* R_2= *find the diseases whose complication is heart disease* *QR= find* R_1 *and* R_2 *whose treatment is antibiotics*	disease whose symptom is fever and disease whose complication is heart disease whose treatment is antibiotics
We have a database of diseases. We would like to first find the incubation period of fever; and then find the diseases which have incubation period longer than fever; these diseases can be also treated with antibiotics.	R_1= *find incubation period of fever* R_2= *find diseases whose incubation period is larger than* R_1 *QR= find* R_2 *whose treatment is antibiotics*	diseases whose incubation period is larger than incubation period of fever whose treatment is antibiotics

Table 10: Templates and artificial sentences shown to AMT crowdworkers together with task description. Examples are taken from the disease domain. *R* and *QR* are shorthands for *Result* and *Query Result*, respectively.

Domain	Tk	Wo	NT	Acc
meeting	16.71	2.83	3.46	0.925
employees	18.64	3.71	3.33	0.938
hotel	16.97	4.08	3.37	0.969
restaurant	17.33	3.72	3.36	0.943
disease	18.95	5.12	3.12	0.925
medication	17.25	3.26	3.15	0.989

Table 11: Average number of tokens (Tk), token overlap between queries and templates (WO), average number of templates (AT) and proportion of parsable templates (Acc) per domain on dynamic mode.

tion and disjunction (Example 2 in Table 10), and when a subquery acts as object of comparison in a longer query (Example 3 in Table 10).

4.4 Data Collection in Dynamic Mode

Next we provide evaluation of our framework in the dynamic mode which we view as an annotation tool for domain managers. Because it is not realistic to recruit a large number of domain managers to participate in our evaluation, we ran this experiment on AMT with crowdworkers who were paid more to compensate for the increased workload. For each domain, workers were given unfilled templates and a table describing naturalized predicates of that domain. They were also given instructions and examples explaining how to use them. Workers were then asked to chose the templates, fill them, and write down a query summarizing the task. We recruited 50 workers for each domain (300 in total) and each was asked to complete two tasks in one HIT worth 1$.

Table 11 shows the statistics of the data we obtained. Workers tend to use more than 3 templates (on average) per task, and the degree of paraphrasing is increased compared to the static mode which suggests that the dynamic mode provides more flexibility for annotators to create data. Moreover, across domains, 90% of the logical forms generated by AMT workers are executable. Although we envisage domain managers as the main users of the dynamic mode, the result indicates that (with proper instructions and examples) naive AMT workers can generate meaningful logical forms to some extent. Inspection of the output failures revealed two common reasons. Firstly, annotators filled in templates with wrong types. For example, one worker filled the LookupKey template with an entity (e.g., *find all* [*annual review*]) instead of a database key; and another worker used Count as the first template followed by Filter, however type constraints require Filter to take a set of entities as argument instead of a number (as returned by Count). Secondly, annotators ignored information in the table and created tasks using non-existing predicates. Since the logical forms can be tested for their validity automatically, providing feedback on the fly may yield a higher percentage of executable queries.

5 Conclusions

This work investigates the lifecycle of semantic parsing for arbitrary domains and degrees of compositionality. Our template-based data collection framework supports two modes (static vs. dynamic) that strike a balance between efficiency and flexibility. Our neural parser leverages the annotations we obtain to generate derivation trees of logical forms. Using the static mode, we built a dataset of six domains and validated the effectiveness of our neural parser. We also experimented with the dynamic mode and released a demo showcasing it as an annotation tool (see supplementary material).

References

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA.

Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. 2017. Learning structured natural language representations for semantic parsing. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 44–55.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China.

Jonathan Herzig and Jonathan Berant. 2017. Neural semantic parsing over multiple knowledge-bases. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 623–628, Vancouver, Canada.

Mohit Iyyer, Wen-tau Yih, and Ming-Wei Chang. 2017. Search-based neural structured learning for sequential question answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1821–1831, Vancouver, Canada.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12–22, Berlin, Germany.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1517–1527.

Tom Kwiatkowksi, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1223–1233, Cambridge, MA.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1512–1523, Edinburgh, Scotland.

Percy Liang, Michael Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 590–599, Portland, Oregon.

Alane Suhr, Srinivasan Iyer, and Yoav Artzi. 2018. Learning to map context-dependent sentences to executable formal queries. ArXiv preprint arXiv:1804.06868.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342, Beijing, China.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. In *Proceedings of 21st Conference in Uncertainilty in Artificial Intelligence*, pages 658–666, Edinburgh, Scotland.